JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 33 (2024) 258–263
Received: 28 May 2024
Accepted: 8 October 2024

Performance analysis of .Net and Spring Boot microservices on Microsoft Azure

Analiza wydajności mikroserwisów .Net i Spring Boot na platformie Microsoft Azure

Konrad Krekora*

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

This article compares the efficiency of microservices written in Java and C#, using Spring Cloud for Java and Consul for C#. The measurements were carried out in three stages, such as basic service implementation, application of design patterns, and deployment on the Microsoft Azure cloud platform. The results illustrate that Java microservices are generally more efficient, although that difference diminishes on the cloud platform where .Net shows better optimization with Microsoft Azure.

Keywords: microservice; Java; C#; cloud platform

Streszczenie

W tym artykule porównano wydajność mikroserwisów napisanych w językach Java i C#, wykorzystując Spring Cloud dla Javy oraz Consul dla C#. Pomiary przeprowadzono w trzech etapach: podstawowa implementacja serwisów, zastosowanie wzorców projektowych oraz wdrożenie na platformie chmurowej Microsoft Azure. Wyniki pokazują, że mikroserwisy Java są generalnie wydajniejsze, choć różnica ta maleje na platformie chmurowej, gdzie .Net wykazuje lepszą optymalizację z Microsoft Azure.

Słowa kluczowe: mikroserwisy; Java; C#; platforma chmurowa

*Corresponding author

Email address: **s96757@pollub.edu.pl** (K. Krekora)

Published under Creative Common License (CC BY 4.0 Int.)

1. Wstęp

Współczesne oprogramowanie musi sprostać coraz wyższym wymaganiom w zakresie wydajności, elastyczności i niezawodności. W odpowiedzi na te wyzwania, rozwijane są nowe technologie i koncepcje, takie jak mikroserwisy, które oferują liczne korzyści w porównaniu do tradycyjnych aplikacji monolitycznych. Dzięki swoim zaletom mikroserwisy zyskały popularność jako rozwiązanie umożliwiające efektywniejsze zarządzanie złożonymi aplikacjami.

W niniejszym artykule dokonano porównania technologii mikroserwisowych w językach Java i C#, uwzględniając ich zalety, wady oraz różnice w wydajności. Przedstawiono również kluczowe komponenty architektury mikroserwisowej, takie jak wykrywanie usług (ang. service discovery) i równoważenie obciążenia (ang. load balancing), oraz ich wpływ na wydajność mikroserwisów.

2. Cel i zakres badań

Celem badań jest przeprowadzenie czasowej analizy porównawczej technologii mikroserwisowych w językach programowania Java i C#. Badania ukierunkowane są na przedstawienie zalet i możliwości rozwiązań mikroserwisowych oraz ukazanie ich wpływu na wydajność aplikacji w obu językach. Głównym zadaniem jest wykazanie, że Java, mimo niegdyś powszechnej opinii o jej niższej wydajności, w kontekście mikroserwisów jest równie dobrym wyborem jak C#.

Zakres badań obejmuje przegląd literatury na temat mikroserwisów, wybór odpowiednich technologii dla języków Java i C# oraz stworzenie dwóch aplikacji symulujących grę planszową w celu przeprowadzenia czasowej analizy porównawczej.

3. Oprogramowania mikroserwisowe

Nie istnieje jeden uniwersalny przepis na tworzenie aplikacji mikroserwisowych, zarówno w ogólnym kontekście, jak i w ramach specyficznych języków programowania. Architektura mikroserwisowa dzieli aplikacje na niewielkie, niezależne serwisy, z których każdy może być rozwijany, wdrażany oraz skalowany niezależnie od reszty systemu. Języki, takie jak Java i C#, oferują wiele rozwiązań wspomagających tworzenie mikroserwisów, które mogą się różnić nawet w obrębie jednego języka [1]. Na przykład, mikroserwisy w Javie mogą być tworzone przy użyciu narzędzi Spring Cloud lub Java Enterprise MicroProfile [2]. C# oferuje takie narzędzia jak ASP.NET Core, wspierające budowanie skalowalnych aplikacji mikroserwisowych.

Warto zauważyć, że niektóre rozwiązania są uniwersalne i niezależne od używanego języka programowania. Przykładami takich rozwiązań są brokerzy wiadomości, tacy jak RabbitMQ czy Apache Kafka. Narzędzia te umożliwiają asynchroniczną komunikację między mikroserwisami, co jest kluczowe dla zapewnienia wysokiej wydajności oraz skalowalności systemów. Dzięki temu, niezależnie od tego, czy mikroserwisy są napisane

w Javie, C#, czy innym języku, mogą efektywnie współpracować i wymieniać dane [3].

3.1. Service Discovery i Load Balancing

Mikroserwisy składają się z wielu elementów, z których jednym z najważniejszych jest wykrywanie usług. Jest to mechanizm rejestracji i odnajdywania serwisów, umożliwiający im komunikację bez potrzeby ręcznego konfigurowania adresów IP. W Javie popularnym narzędziem jest Eureka z ekosystemu Spring Cloud, natomiast w C# często używa się Consul autorstwa HashiCorp [4, 5].

Oba te narzędzia integrują się z mechanizmami równoważenia obciążenia, które rozdzielają przychodzące żądania między aktywne instancje serwisu, zapewniając równomierne obciążenie [6]. Zatem wykrywanie usług i równoważenie obciążenia są kluczowe dla dynamiczności i elastyczności architektury mikroserwisowej, ułatwiając zarządzanie i skalowanie dużych systemów [7].

3.2. Pozostałe składowe mikroserwisu

W skład mikroserwisów wchodzi wiele innych kluczowych komponentów. Jednym z nich jest API Gateway, który pełni rolę punktu wejścia dla żądań klientów, kierując je następnie do odpowiednich mikrousług. Dodatkowo przeprowadza on uwierzytelnianie, autoryzację oraz inne zadania, takie jak zapisywanie w pamięci podręcznej (ang. cache). Przykładami są ZUUL autorstwa Netflix w przypadku Javy oraz Ocelot w C# [8].

Kolejnym ważnym elementem jest bezpiecznik (ang. circuit breaker). Jest to wzorzec projektowy, który monitoruje awarie usług i umożliwia systemowi dalsze działanie, nawet w przypadku, gdy powiązana usługa przestanie odpowiadać. W Javie narzędziem do implementacji tego wzorca może być Hystrix, natomiast w C# często używa się Polly [9].

Dzięki elastyczności konfiguracji i różnorodnym wymaganiom poszczególne mikroserwisy mogą korzystać z różnych zestawów komponentów, dostosowanych do specyficznych potrzeb projektu. To pozwala na optymalne wykorzystanie zasobów i lepsze dostosowanie do zmieniających się warunków operacyjnych [10].

4. Metoda badań

Na potrzeby analizy porównawczej zostały utworzone dwie aplikacje mikroserwisowe w językach programowania C# i Java.

Aplikacje stworzone w obu językach posiadają trzy typy mikroserwisów:

- Board Service – mikroserwis obsługujący planszę dla graczy. Zawiera jeden punkt końcowy przyjmujący parametry testowe, takie jak liczba gier, które mają się wykonać.
- Player Service – mikroserwis obsługujący graczy. Zawiera punkty końcowe obsługujące tworzenie, pobieranie listy oraz usuwanie graczy. Dane o graczach są przechowywane w bazie danych.
- Soldier Service – mikroserwis obsługujący żołnierzy graczy. Zawiera punkty końcowe obsługujące tworzenie, ruch po planszy, walkę, pobieranie listy aktywnych oraz usuwanie poległych żołnierzy z planszy. Dane o żołnierzach są przechowywane w bazie danych.

Gra polega na utworzeniu baz dwóch graczy na planszy o wymiarach 10 na 10 pól. W każdej turze gracz otrzymuje nowych żołnierzy, którzy następnie poruszają się losowo po planszy i walczą z żołnierzami drugiego gracza. W momencie gdy jednostka jednego z graczy stanie obok bazy drugiego, wtedy drugi gracz przegrywa, a gra się kończy.

Badania polegają na pomiarze łącznego czasu 100 gier dla każdego z pięciu poziomów współczynnika odradzania (ang. spawn rate). Im wyższy współczynnik odradzania, tym więcej żołnierzy pojawia się w każdej turze. Przebieg gry obejmuje operacje, które polegają na komunikacji między mikroserwisami, przetwarzaniu danych w ramach mikroserwisu oraz zapisie przetworzonych danych do bazy.

4.1. Punkty końcowe

Logika mikroserwisów w obu językach jest taka sama, a implementacja niemal identyczna (Listingi 1-2). Choć obie technologie mają swoje unikalne cechy i specyficzne narzędzia, podstawowe zasady i wzorce projektowe są w dużej mierze uniwersalne. Jak widać na listingach, struktura i sposób tworzenia punktów końcowych (ang. endpoints) w obu językach jest bardzo podobna. Oba przykłady definiują kontroler HTTP, który obsługuje żądania GET i zwraca odpowiedź.

Listing 1: Kod kontrolera głównego mikroserwisu w Javie

```
@RestController
@AllArgsConstructor
@RequestMapping("/api/board")
public class BoardController {

    private BoardService boardService;

    new *
    @GetMapping("/getBoard/{spawnRate}/{testsQuantity}")
    public GameData getBoard(
            @PathVariable("spawnRate") int spawnRate,
            @PathVariable("testsQuantity") int testsQuantity) {
        return boardService.getBoard(spawnRate, testsQuantity);
    }
```

Listing 2: Kod kontrolera głównego mikroserwisu w C#

```
[ApiController]
[Route("[controller]")]
1 odwołanie
public class BoardController : ControllerBase
{
    private readonly BoardServices boardServices;

    Odwołania: 0
    public BoardController(BoardServices boardServices)
    {
        this.boardServices = boardServices;
    }

    [HttpGet]
    [Route("getBoard/{spawnR}/{testsQ}")]
    [EnableCors]
    Odwołania: 0
    public async Task<GameData> GetBoard(int spawnR, int testsQ) {
        return await boardServices.getBoard(spawnR, testsQ);
    }
```

4.2. Hipoteza badawcza

Mikroserwisy utworzone w oparciu o Spring Cloud są wydajniejsze czasowo od tych stworzonych w technologii .Net.

4.3. Scenariusz badawczy

Scenariusz badawczy zakłada:

1. Wysłanie żądania do punktu końcowego getBoard w kontrolerze mikroserwisu Board Service, a następnie wywołanie metody o tej samej nazwie w serwisie.
2. Rozpoczęcie zliczania czasu oraz liczby tur, jakie upłyną w trakcie wykonywania aplikacji testowej.
3. Otwarcie pętli for, w której aplikacja testowa przeprowadzi grę sto razy. Operacje składające się na przeprowadzenie jednej gry:
   a) Rozpoczęcie pętli do while wykonującej tury graczy tak długo, aż flaga gameOverFlag oznaczająca zakończenie gry nie zostanie ustawiona na 1,
   b) Utworzenie nowych graczy, jeśli flaga gameStartedFlag jest równa 0:
      i. wysłanie żądania do REST API mikroserwisu z graczami o utworzenie dwóch nowych graczy,
      ii. przyjęcie żądania przez mikrousługę graczy,
      iii. wygenerowanie dwóch nowych graczy,
      iv. zapisanie nowo utworzonych graczy w bazie danych,
      v. zwrócenie listy graczy w odpowiedzi na żądanie,
      vi. zmiana flagi gameStartedFlag na 1,
      vii. ustawienie flagi gameOverFlag na 0,
   c) Przełączenie tury gracza oraz utworzenie żołnierzy dla tego gracza:
      i. wywołanie metody switchPlayerTurn ustawiającej pierwszego gracza w przypadku pierwszej tury lub zmieniającej turę gracza w dalszej części gry,
      ii. wysłanie żądania z informacjami o aktualnym graczu do REST API mikroserwisu z żołnierzami o wygenerowanie dla niego nowych żołnierzy na jego współrzędnych,
      iii. przyjęcie żądania przez mikrousługę żołnierzy i wywołanie metody spawnSoldiers w serwisie,
      iv. wygenerowanie nowych żołnierzy w liczbie zależnej od ustawionej częstotliwości tworzenia w zakresie od 1 do 5 (taka sama liczba żołnierzy dla obu graczy),
      v. zapisanie żołnierzy do bazy danych,
      vi. zwrócenie informacji do mikroserwisu planszy o poprawnym utworzeniu żołnierzy,
   d) Wykonanie ruchu przez żołnierzy aktualnego gracza:
      i. wysłanie żądania z id aktualnego gracza do REST API mikroserwisu z żołnierzami o wykonanie ruchu jego żołnierzy,
      ii. przyjęcie żądania przez kontroler mikrousługi żołnierzy i wywołanie metody moveSoldiers w serwisie,
      iii. wykonanie zapytania do bazy danych o pobranie wszystkich żołnierzy dla danego gracza,
      iv. wygenerowanie nowych pozycji dla żołnierzy,
      v. wykonanie zapytania do bazy danych zapisującego listę żołnierzy aktywnego gracza z ich nowymi pozycjami na planszy,
      vi. zwrócenie informacji do mikroserwisu planszy o poprawnym wykonaniu ruchu przez żołnierzy,
   e) Wykonanie ataku przez żołnierzy aktualnego gracza:
      i. wysłanie żądania z id obu graczy do REST API mikroserwisu z żołnierzami o wykonanie ataku żołnierzy aktywnego gracza na przeciwniku,
      ii. przyjęcie żądania przez kontroler mikrousługi żołnierzy i wywołanie metody soldiersFight w serwisie,
      iii. wykonanie zapytania do bazy danych o pobranie wszystkich żołnierzy dla aktywnego gracza,
      iv. wykonanie zapytania do bazy danych o pobranie wszystkich żołnierzy dla gracza przeciwnika,
      v. porównanie współrzędnych jednostek na planszy i dodanie ich do listy wszystkich żołnierzy przeciwnika znajdujących się w sąsiedztwie jednostek aktywnego gracza,
      vi. wykonanie zapytania do bazy danych o usunięcie żołnierzy przeciwnika, którzy znajdowali się na utworzonej liście,
      vii. jeżeli baza przeciwnika również znalazła się w sąsiedztwie żołnierzy aktualnego gracza, to metoda wywoła zapytanie do REST API usługi graczy o usunięcie danego gracza z bazy danych, wykona zapytanie o usunięcie z własnej bazy danych żołnierzy gracza o tym id oraz zwróci do mikroserwisu planszy id pokonanego gracza, w innym przypadku zwróci 0,
   f) W przypadku gdy poprzednie żądanie do mikroserwisu żołnierzy zwróciło id gracza, wtedy zostaje uruchomiona procedura zakończenia gry:
      i. wysłanie żądania do REST API mikroserwisu graczy o usunięcie gracza z bazy danych,
      ii. wywołanie metody removePlayer w serwisie, która wykonuje zapytanie do bazy danych o usunięcie gracza,
      iii. zwrócenie do usługi planszy informacji z id usuniętego gracza,
      iv. ustawienie flagi gameOverFlag na 1,
      v. koniec gry, zakończenie pętli do while,
   g) W przypadku gdy żądanie do mikroserwisu żołnierzy o przeprowadzenie ataku nie zwróci id gracza, tylko 0, wtedy jednostki zostaną rozmieszczone na planszy i rozpocznie się kolejna tura.
4. Po przeprowadzeniu 100 gier końcówka REST zwróci uzyskane wyniki jako obiekt GameData w formacie JSON.

5. Wyniki pomiarów

W tym rozdziale przedstawiono wyniki dotyczące mikroserwisów nieskonteneryzowanych na trzech etapach tworzenia. Pierwszy etap obejmuje podstawowe mikrousługi bez wykorzystania wzorców projektowych, takich jak wykrywanie usług czy równoważenie obciążenia. W drugim etapie mikroserwisy są wzbogacone o wyżej wymienione wzorce. Ostatni etap koncentruje się na przetestowaniu wydajności mikroserwisów osadzonych na platformie chmurowej Microsoft Azure, z wykorzystaniem zasobów, takich jak Azure Spring Apps i Azure App Services.

5.1. Podstawowe mikroserwisy

Pierwsze testy zostały przeprowadzone na mikroserwisach nieskonteneryzowanych bez wykorzystania dodatkowych wzorców projektowych, takich jak service discovery lub load balance.

Wszystkie testy przedstawiają pomiary dla 100 gier w językach programowania Java i C#. Pomiary wykonano dla pięciu grup badawczych ze współczynnikiem odradzania żołnierzy w zakresie od jednego do pięciu. Wyższy współczynnik oznacza dodanie większej liczby żołnierzy dla każdego gracza w każdej turze.

Tabela 1 przedstawia łączny czas potrzebny na wykonanie 100 gier dla mikroserwisów bez wzorców projektowych.

Tabela 1: Łączny czas [ms] potrzebny na wykonanie 100 gier

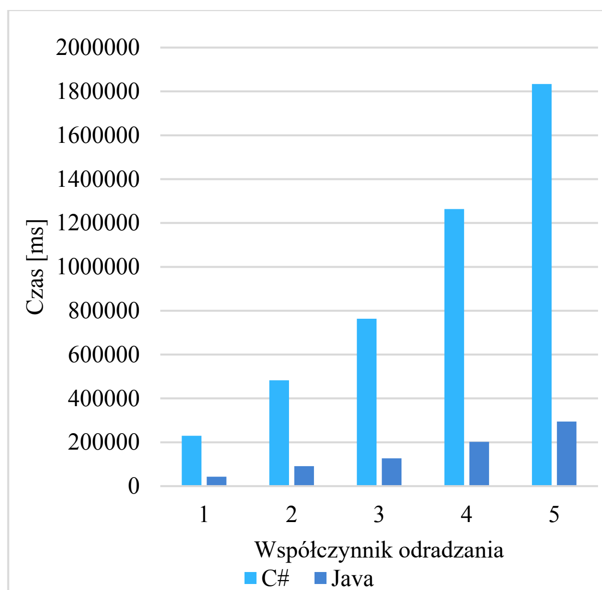
Współczynnik odradzania	C# [ms]	Java [ms]
1	224621	58926
2	500707	144932
3	763750	204899
4	1263677	280277
5	1833738	502619

Tabela 2 przedstawia liczbę tur potrzebnych na wykonanie 100 gier dla mikroserwisów bez wzorców projektowych.

Tabela 2: Łączna liczba tur podczas 100 gier

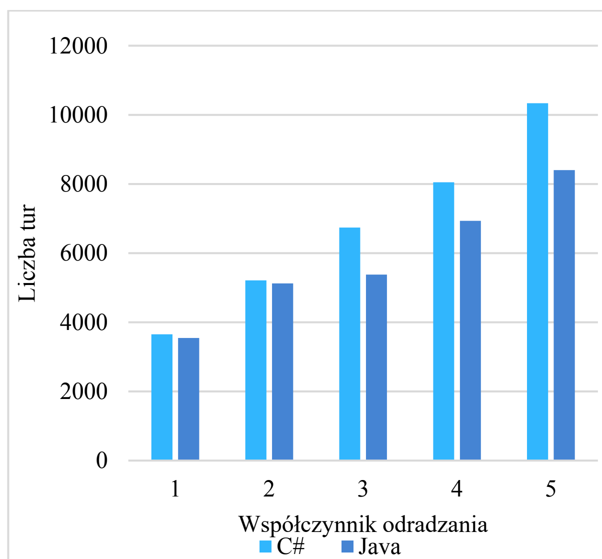
Współczynnik odradzania	C#	Java
1	3930	4056
2	5458	5796
3	6740	6415
4	8549	6977
5	10338	9502

Po przeprowadzonych pomiarach można zaobserwować wyraźną różnicę w wydajności czasowej między mikroserwisem utworzonym w Javie a tym utworzonym w C# (Rysunek 1). Różnica utrzymuje się na stosunkowo równym poziomie dla wszystkich grup pomiarowych. Wyniki otrzymane dla podstawowych mikroserwisów zgadzają się z hipotezą badawczą.



Rysunek 1: Czas [ms] w zależności od współczynnika odradzania.

Pomiar liczby tur został przeprowadzony w charakterze kontrolnym (Rysunek 2). Im mniejsze różnice między słupkami, tym dokładniejsze wyniki z racji na podobne obciążenie sprzętu w przypadku obu języków. Niewielkie różnice między słupkami świadczą o poprawnej implementacji logiki w aplikacjach, która jest identyczna dla obydwu języków.



Rysunek 2: Liczba tur w zależności od współczynnika odradzania.

5.2. Podstawowe mikroserwisy

Drugi etap testów został przeprowadzony na mikrousługach wykorzystujących wzorce projektowe, takie jak wykrywanie usług, brama API oraz równoważenie obciążenia.

Tabela 3 przedstawia łączny czas potrzebny na wykonanie 100 gier dla mikroserwisów wykorzystujących dodatkowe wzorce projektowe.

Tabela 3: Łączny czas [ms] potrzebny na wykonanie 100 gier

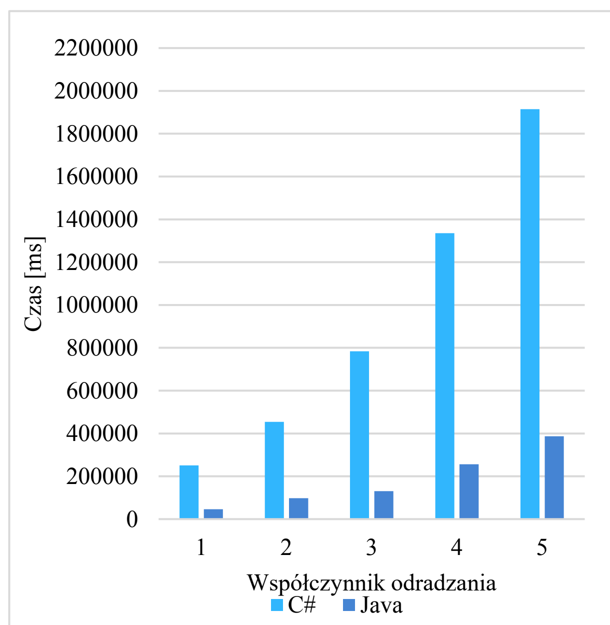
Współczynnik odradzania	C# [ms]	Java [ms]
1	251174	46308
2	454454	97620
3	783530	130548
4	1335172	256307
5	1914557	387322

Tabela 4 przedstawia liczbę tur potrzebnych na wykonanie 100 gier dla mikroserwisów wykorzystujących dodatkowe wzorce projektowe.

Tabela 4: Łączna liczba tur podczas 100 gier

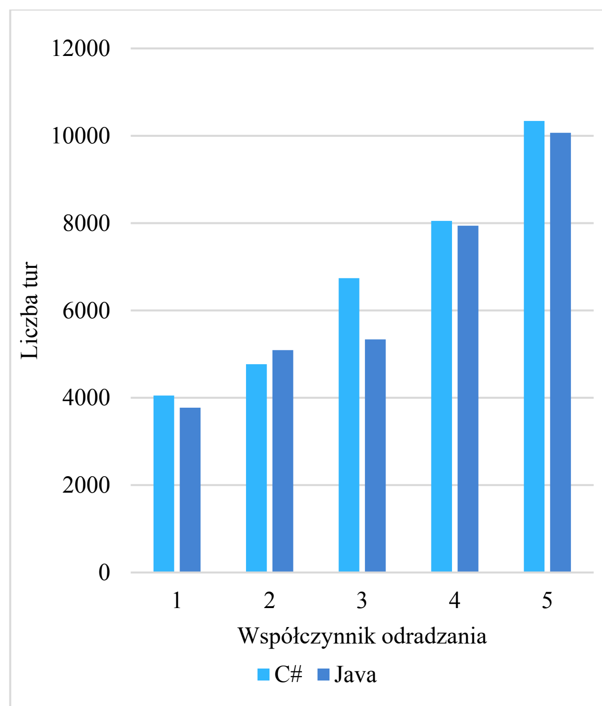
Współczynnik odradzania	C#	Java
1	4050	3773
2	4770	5092
3	6740	5337
4	8049	7942
5	10338	10070

Po przeprowadzonych pomiarach można zaobserwować utrzymującą się wyraźną różnicę w wydajności czasowej między mikroserwisem utworzonym w Javie a tym utworzonym w C# (Rysunek 3). Różnica utrzymuje się na stosunkowo równym poziomie dla wszystkich grup pomiarowych. Wyniki otrzymane dla mikroserwisów wykorzystujących wzorce projektowe również zgadzają się z hipotezą badawczą.



Rysunek 3: Czas [ms] w zależności od współczynnika odradzania.

Pomiar liczby tur został przeprowadzony w charakterze kontrolnym (Rysunek 4). Im mniejsze różnice między słupkami, tym dokładniejsze wyniki z racji na podobne obciążenie sprzętu w przypadku obu języków. Niewielkie różnice między słupkami świadczą o poprawnej implementacji logiki w aplikacjach, która jest identyczna dla obydwu języków.



Rysunek 4: Liczba tur w zależności od współczynnika odradzania.

5.3. Mikroserwisy na platformie chmurowej

Trzeci etap testów został przeprowadzony na mikrousługach osadzonych na platformie chmurowej Microsoft Azure z wykorzystaniem zasobów dostępnych w jej ofercie. Zasobem określana jest każda dostępna usługa, z której można skorzystać na tej platformie chmurowej.

Pierwszym z wykorzystanych zasobów jest Azure Spring Apps. Jest to usługa dedykowana dla mikroserwisów utworzonych z pomocą zestawu narzędzi dostępnych w ramach Spring Cloud. Wspiera ona m.in. wykrywanie usług z pomocą Netflix Eureka.

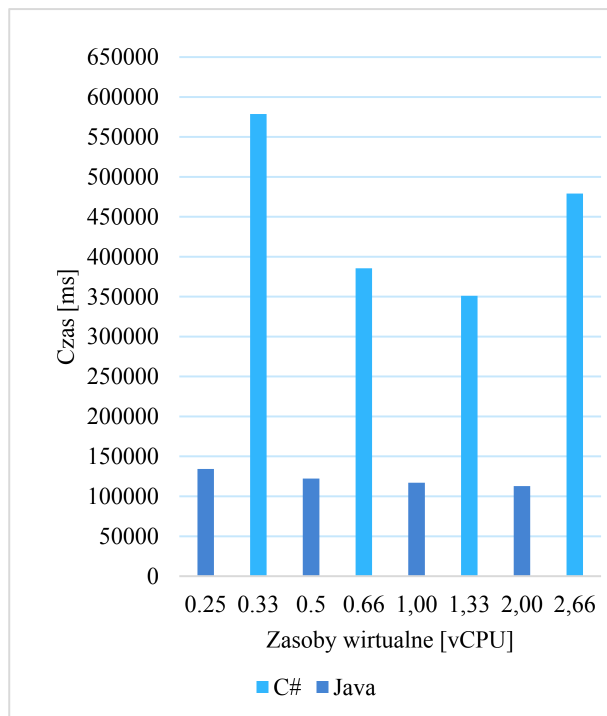
Drugim z wykorzystanych zasobów jest Azure App Services. Jest to podstawowy zasób wykorzystywany do aplikacji webowych, który świetnie nadaje się też do wdrażania prostych mikroserwisów.

Tabela 5 przedstawia łączny czas potrzebny na wykonanie 100 gier dla mikroserwisów osadzonych na platformie chmurowej. Pomiary wykonano dla wybranych wartości zasobów wirtualnych [vCPU] ze stałym współczynnikiem odradzania żołnierzy ustawionym na 1. Ze względu na różnice w dystrybucji (w Azure Spring Apps przydziela się zasób na każdą usługę oddzielnie; w Azure App Services na wszystkie usługi razem) występują różnice w ilości przypisanych zasobów wirtualnych pomiędzy C# a Java.

Tabela 5: Łączny czas [ms] potrzebny do wykonania 100 gier w zależności od zasobów przypadających na jeden mikroserwis [vCPU]

Przypisane zasoby wirtualne [vCPU]	C# [ms]	Java [ms]
0.25	-	134200
0.33	578682	-
0.5	-	122155
0.66	385467	-
1	-	117013
1.33	351021	-
2	-	112841
2.66	479161	-

Po przeprowadzonych pomiarach na platformie chmurowej można zaobserwować znaczącą różnicę względem pomiarów w środowisku lokalnym. Czas działania obu mikroserwisów jest znacząco dłuższy, jednak dysproporcja wyników jest znacznie mniejsza. Mimo to różnica w wydajności między mikroserwisami stworzonymi w języku Java a tymi stworzonymi w C# wciąż jest duża (Rysunek 5).



Rysunek 5: Łączny czas [ms] potrzebny do wykonania 100 gier w zależności od zasobów przypadających na jeden mikroserwis [vCPU].

6. Wnioski

Mikroserwisy są przydatnym rozwiązaniem nie tylko dla dużych, ale również mniejszych projektów. Zarówno C#, jak i Java są odpowiednie do tworzenia aplikacji mikroserwisowych, jednak różnią się wydajnością, łatwością obsługi na Microsoft Azure oraz skalowalnością. W testach lokalnych i na platformie chmurowej Microsoft Azure, mikroserwisy Java okazały się wydajniejsze czasowo, potwierdzając tym samym postawioną hipotezę badawczą. Zwiększenie zasobów wirtualnych bardziej wpływa na działanie mikroserwisów C#, a większy współczynnik zasobu na serwis zmniejsza różnice w wydajności między Javą a C#. Testy wykazały również, że środowisko .NET w Visual Studio oferuje liczne rozszerzenia, które sprawiają, że jest ono lepiej przystosowane do obsługi platformy chmurowej Microsoft Azure.

Literatura

[1] E. Wolff, Microservices: flexible software architecture, Addison-Wesley Professional, 2016.

[2] C. Richardson, Microservices patterns: with examples in Java, Manning publications, 2018.

[3] P. Sbarski, S. Kroonenburg, Serverless architectures on AWS: with examples using Aws Lambda, Manning publications, 2017.

[4] P. Gankiewicz, Mikroserwisy, wyzwania rozproszonej architektury, https://geek.justjoin.it/mikroserwisy-wyzwania-rozproszonej-architektury/, [27.05.2024].

[5] S. Haselböck, R. Weinreich, G. Buchgeher, Decision guidance models for microservices: service discovery and fault tolerance, Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems (2017) 1-10, https://doi.org/10.1145/3123779.3123804.

[6] R. Yu, V. T. Kilari, G. Xue, D. Yang, Load balancing for interdependent IoT microservices, In IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE (2019) 298-306, https://doi.org/10.1109/INFOCOM.2019.8737450.

[7] P. Johansson, Effcient Communication With Microservices, Dissertation, Umeå University, 2017.

[8] R. Xu, W. Jin, D. Kim, Microservice security agent based on API gateway in edge computing, Sensors 19(22) (2019) 4905, https://doi.org/10.3390/s19224905.

[9] F. Montesi, J. Weber, Circuit breakers, discovery, and API gateways in microservices, arXiv (2016), https://doi.org/10.48550/arXiv.1609.05830.

[10] B. Christudas, Practical Microservices Architectural Patterns, Apress, Berkeley, CA (2019) 87-104, https://doi.org/10.1007/978-1-4842-4501-9_5.